

APPENDIX A

FINITE FIELD ARITHMETIC FOR FEC-I AND FEC-II CODES

10. Scope. This appendix is not a mandatory part of the standard. The information contained in it is intended for guidance only.

20. Applicable documents. This section is not applicable to this appendix.

30. Finite field arithmetic for FEC-I and FEC-II codes. The field arithmetic used in these codes is defined by the field generator polynomial $x^8 + x^5 + x^3 + x^2 + 1$. This appendix describes what this means in terms of performing field operations such as addition and multiplication as part of a computation implementing the FEC codes.

There are 256 field elements, each represented by an eight-bit byte. The following elements (given in hexadecimal) have special significance:

- 0x00 --- 0, the additive identity
- 0x01 --- 1, the multiplicative identity
- 0x02 --- α , the field generator

Arithmetic computation for this field can be defined by the following six postulates:

- a. For any elements x and y , $x+y$ is the bitwise exclusive-OR of x and y .
Therefore, $x + 0 = 0 + x = x$; $x + x = 0$; $x + y = y + x$; and for elements x , y and z , $(x+y) + z = x + (y+z)$.
- b. For any element x , $x \times 1 = x$.
- c. Commutative law for multiplication: for any elements x and y , $xy = yx$.
- d. Associative law for multiplication: for any elements x , y and z , $(xy)z = x(yz)$.
- e. Distributive law: for any elements c , x , and y , $c(x+y) = cx + cy$.
- f. The following multiplication table, combined with the distributive law, defines multiplication for the entire field (all values are hexadecimal):

	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
0x01	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
0x02	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x2 D
0x04	0x04	0x08	0x10	0x20	0x40	0x80	0x2D	0x5A
0x08	0x08	0x10	0x20	0x40	0x80	0x2D	0x5A	0xB4
0x10	0x10	0x20	0x40	0x80	0x2D	0x5A	0xB4	0x45
0x20	0x20	0x40	0x80	0x2D	0x5A	0xB4	0x45	0x8A
0x40	0x40	0x80	0x2D	0x5A	0xB4	0x45	0x8A	0x39
0x80	0x80	0x2D	0x5A	0xB4	0x45	0x8A	0x39	0x72

Based on the above, exponentiation of a field element x by a non-negative integer n is defined so that x^0 is 1, and x^n is the result of multiplying x together n times. The field element α is said to generate the field because α^0 through α^{254} are distinct, and are the 255 non-zero field elements.

The inverse of a field element α^i is α^{255-i} : $\alpha^i \alpha^{255-i} = 1$ for any integer i between 0 and 254. Exponentiation by negative integer exponents is defined such that x^{-1} is the inverse of x .

Example: multiply 0x80 by 0x06, using the multiplication table and the distributive law:

$$\begin{aligned}
 & 0x80 \times 0x06 \\
 &= 0x80(0x04 + 0x02) \\
 &= 0x80 \times 0x04 + 0x80 \times 0x02 \\
 &= 0x5A + 0x2D \\
 &= 0x77.
 \end{aligned}$$

Example: compute the value of $\alpha^3 + \alpha^{65}$

$$\begin{aligned}
 \alpha^3 &= 0x08 \\
 \alpha^{65} &= 0xC1 \\
 \alpha^3 + \alpha^{65} &= 0xC9
 \end{aligned}$$

The 256 field elements are provided in Table A-1.

Table A-I. Field Elements.

i	a ⁱ
0	01
1	02
2	04
3	08
4	10
5	20
6	40
7	80
8	2D
9	5A
10	B4
11	45
12	8A
13	39
14	72
15	E4
16	E5
17	E7
18	E3
19	EB

i	a ⁱ
52	95
53	07
54	0E
55	1C
56	38
57	70
58	E0
59	ED
60	F7
61	C3
62	AB
63	7B
64	F6
65	C1
66	AF
67	73
68	E6
69	E1
70	EF
71	F3

i	a ⁱ
104	4E
105	9C
106	15
107	2A
108	54
109	A8
110	7D
111	FA
112	D9
113	9F
114	13
115	26
116	4C
117	98
118	1D
119	3A
120	74
121	E8
122	FD
123	D7

i	a ⁱ
156	F5
157	C7
158	A3
159	6B
160	D6
161	81
162	2F
163	5E
164	BC
165	55
166	AA
167	79
168	F2
169	C9
170	BF
171	53
172	A6
173	61
174	C2
175	A9

i	a ⁱ
208	DE
209	91
210	0F
211	1E
212	3C
213	78
214	F0
215	CD
216	B7
217	43
218	86
219	21
220	42
221	84
222	25
223	4A
224	94
225	05
226	0A
227	14

i	a ⁱ	i	a ⁱ	i	a ⁱ	i	a ⁱ	i	a ⁱ
20	FB	72	CB	124	83	176	7F	228	28
21	DB	73	BB	125	2B	177	FE	229	50
22	9B	74	5B	126	56	178	D1	230	A0
23	1B	75	B6	127	AC	179	8F	231	6D
24	36	76	41	128	75	180	33	232	DA
25	6C	77	82	129	EA	181	66	233	99
26	D8	78	29	130	F9	182	CC	234	1F
27	9D	79	52	131	DF	183	B5	235	3E
28	17	80	A4	132	93	184	47	236	7C
29	2E	81	65	133	0B	185	8E	237	F8
30	5C	82	CA	134	16	186	31	238	DD
31	B8	83	B9	135	2C	187	62	239	97
32	5D	84	5F	136	58	188	C4	240	03
33	BA	85	BE	137	B0	189	A5	241	06
34	59	86	51	138	4D	190	67	242	0C
35	B2	87	A2	139	9A	191	CE	243	18
36	49	88	69	140	19	192	B1	244	30
37	92	89	D2	141	32	193	4F	245	60
38	09	90	89	142	64	194	9E	246	C0
39	12	91	3F	143	C8	195	11	247	AD
40	24	92	7E	144	BD	196	22	248	77

TABLE A-I. Field Elements - Continued.

i	a ⁱ	i	a ⁱ	i	a ⁱ	i	a ⁱ	i	a ⁱ
41	48	93	FC	145	57	197	44	249	EE
42	90	94	D5	146	AE	198	88	250	F1
43	0D	95	87	147	71	199	3D	251	CF
44	1A	96	23	148	E2	200	7A	252	B3
45	34	97	46	149	E9	201	F4	253	4B
46	68	98	8C	150	FF	202	C5	254	96
47	D0	99	35	151	D3	203	A7	255	00
48	8D	100	6A	152	8B	204	63		
49	37	101	D4	153	3B	205	C6		
50	6E	102	85	154	76	206	A1		
51	DC	103	27	155	EC	207	6F		

APPENDIX B

EXAMPLE SOFTWARE FOR FEC-I

10. Scope. This appendix is not a mandatory part of the standard. The information contained in it is intended for guidance only.

20. Applicable documents. This section is not applicable to this appendix.

30. Example software for FEC-I.

Contents of the file "ReadMe" are as follows:

- a. This directory contains three files:
 - this file ("ReadMe")
 - "x3b11", which has log/alog tables for the finite field
 - sample.c, which contains sample encode() and decode() functions for the FEC-I code

When compiled and executed, sample.c performs a sample decoding and should produce the following output:

```
remainder: 33
remainder: 127
remainder: 74
remainder: 208
remainder: 239
remainder: 95
remainder: 43
remainder: 181
remainder: 41
remainder: 150
degW 5
d: 0

dtmp at end of chien(): 0

length: 152
degree (should be 5): 5
uncorrected errors (should be 0): 0
```

Contents of the file "x3b11" are as follows:

256

```
00 00 01 f0 02 e1 f1 35 03 26 e2 85 f2 2b 36 d2
04 c3 27 72 e3 6a 86 1c f3 8c 2c 17 37 76 d3 ea
05 db c4 60 28 de 73 67 e4 4e 6b 7d 87 08 1d a2
f4 ba 8d b4 2d 63 18 31 38 0d 77 99 d4 c7 eb 5b
06 4c dc d9 c5 0b 61 b8 29 24 df fd 74 8a 68 c1
e5 56 4f ab 6c a5 7e 91 88 22 09 4a 1e 20 a3 54
f5 ad bb cc 8e 51 b5 be 2e 58 64 9f 19 e7 32 cf
39 93 0e 43 78 80 9a f8 d5 a7 c8 3f ec 6e 5c b0
07 a1 4d 7c dd 66 da 5f c6 5a 0c 98 62 30 b9 b3
2a d1 25 84 e0 34 fe ef 75 e9 8b 16 69 1b c2 71
e6 ce 57 9e 50 bd ac cb 6d af a6 3e 7f f7 92 42
89 c0 23 fc 0a b7 4b d8 1f 53 21 49 a4 90 55 aa
f6 41 ae 3d bc ca cd 9d 8f a9 52 48 b6 d7 bf fb
2f b2 59 97 65 5e a0 7b 1a 70 e8 15 33 ee d0 83
3a 45 94 12 0f 10 44 11 79 95 81 13 9b 3b f9 46
d6 fa a8 47 c9 9c 40 3c ed 82 6f 14 5d 7a b1 96
```

```
01 02 04 08 10 20 40 80 2d 5a b4 45 8a 39 72 e4
e5 e7 e3 eb fb db 9b 1b 36 6c d8 9d 17 2e 5c b8
5d ba 59 b2 49 92 09 12 24 48 90 0d 1a 34 68 d0
8d 37 6e dc 95 07 0e 1c 38 70 e0 ed f7 c3 ab 7b
f6 c1 af 73 e6 e1 ef f3 cb bb 5b b6 41 82 29 52
a4 65 ca b9 5f be 51 a2 69 d2 89 3f 7e fc d5 87
23 46 8c 35 6a d4 85 27 4e 9c 15 2a 54 a8 7d fa
d9 9f 13 26 4c 98 1d 3a 74 e8 fd d7 83 2b 56 ac
75 ea f9 df 93 0b 16 2c 58 b0 4d 9a 19 32 64 c8
bd 57 ae 71 e2 e9 ff d3 8b 3b 76 ec f5 c7 a3 6b
d6 81 2f 5e bc 55 aa 79 f2 c9 bf 53 a6 61 c2 a9
7f fe d1 8f 33 66 cc b5 47 8e 31 62 c4 a5 67 ce
b1 4f 9e 11 22 44 88 3d 7a f4 c5 a7 63 c6 a1 6f
de 91 0f 1e 3c 78 f0 cd b7 43 86 21 42 84 25 4a
94 05 0a 14 28 50 a0 6d da 99 1f 3e 7c f8 dd 97
03 06 0c 18 30 60 c0 ad 77 ee f1 cf b3 4b 96 00
```

Contents of the file "sample.c" are as follows:

```
* sample version of FEC-I encode and decode routines --
functional for unencoded packet lengths up to 152 bytes */
```

```
#include <stdio.h>
```

```

#define K 152
#define GFCALCFILE "x3b11"

/* array of constants used by Welch-Berlekamp iteration */

unsigned char x[10][12] = {

{ 0x01, 0x01 },
{ 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x2d, 0x5a, 0xb4, 0x45 },
{ 0x01, 0x04, 0x10, 0x40, 0x2d, 0xb4, 0x8a, 0x72, 0xe5, 0xe3, 0xfb, 0x9b },
{ 0x01, 0x08, 0x40, 0x5a, 0x8a, 0xe4, 0xe3, 0xdb, 0x36, 0x9d, 0x5c, 0xba },
{ 0x01, 0x10, 0x2d, 0x8a, 0xe5, 0xfb, 0x36, 0x17, 0x5d, 0x49, 0x24, 0x1a },
{ 0x01, 0x20, 0xb4, 0xe4, 0xfb, 0x6c, 0x5c, 0xb2, 0x24, 0x34, 0x6e, 0x1c },
{ 0x01, 0x40, 0x8a, 0xe3, 0x36, 0x5c, 0x49, 0x90, 0x8d, 0x0e, 0xf7, 0xaf },
{ 0x01, 0x80, 0x72, 0xdb, 0x17, 0xb2, 0x90, 0x37, 0x38, 0x7b, 0xef, 0x82 },
{ 0x01, 0x2d, 0xe5, 0x36, 0x5d, 0x24, 0x8d, 0x38, 0xf6, 0xcb, 0xa4, 0x69 },
{ 0x01, 0x5a, 0xe3, 0x9d, 0x49, 0x34, 0x0e, 0x7b, 0xcb, 0x65, 0x89, 0x35 }
};

```

```

/* packet structure -- maximum lengths:
   netblt packet data field: 100 bytes
   ip datagram: 152 bytes
   encoded packet: 162 bytes (1 RS codeword)
   structures sized for 700 byte packets - later */

```

```

struct packet {
    short length;
    unsigned char data[802];
};

```

```

unsigned short log[256], alog[256];
unsigned char w[6], n[6], m[6], v[6];
unsigned char loc[6], val[6];
short NumErrs;
short d,degW;
unsigned char a,b,s;

```

```

/* main() first initializes log and alog tables */

```

```

main()
{
    struct packet pack;
    int x,i;
}

```

```

scan_gfcalc(); /* function to read in log, alog tables */

/* test code creating a packet with five errors */

pack.length = 162;
for (i=0;i<162;i++) {
    pack.data[i] = (i>4) ? 0 : i+1;
}

/* decode the packet */

decode(&pack);

/* now print some more stuff to verify successful decode */

printf("length: %d\n",pack.length);
d=0;
for (i=0;i<152;i++) if (pack.data[i]) d++;
printf("degree (should be 5): %d\n",degW);
printf("uncorrected errors (should be 0): %d\n",d);

}

scan_gfcalc()
{
FILE *fp; int i,j;
if ((fp=fopen(GFCALCFILE,"r"))==NULL)
    {(void)fprintf(stderr,"bad file\n"); exit(1);}
(void) fscanf(fp,"%d",&j); /* dummy scan */
/* printf("%d\n",j); */

for (i=0;i<256;i++)
{ (void) fscanf(fp,"%x",&j);
  /* printf("%d\n",j); */
  log[i] = (unsigned short) j; }
for (i=0;i<256;i++)
{ (void) fscanf(fp,"%x",&j); alog[i] = (unsigned short) j; }
(void) fclose(fp);
}

unsigned short
invert(x)
unsigned short x;
{
return(alog[(255 - log[x]) % 255]);
}

```

```

}

unsigned short
mult(x,y)
unsigned short x,y;
{
return(x && y ? alog[(log[x] + log[y]) % 255] : 0);
}

/* function to encode a single packet */

encode(p)
struct packet *p;
{
int l,i,j;
if (!(l = p->length)) return;
if (l>152) return; /* later -- encode longer packets */

/* initialize check bytes to zero */

for (j=0;j<10;j++) p->data[l+j] = 0;

/* encode by computing each check byte. i and j have the same
definition as in the equation defining the check bytes in 5.1 */

for (j=0;j<10;j++)
  for (i=10;i<l+10;i++)
    p->data[l+j] ^= mult(p->data[l+9-i],invert(alog[i]^alog[j]));

/* increase packet length by 10 */

p->length += 10;
}

/* function to decode a single packet: first re-encode, then
cal wb(), chien(), correct() */

decode(p)
struct packet *p;
{
int l,i,j;
if (!(l = (p->length)-10)) return;
if (l>152) return; /* later -- decode longer packets */

/* re-encode each check byte -- this is the same as encoding, but

```

without zeroing the check bytes first. i and j have the same definition as in the equation defining encoding checks in 5.1 */

```
for (j=0;j<10;j++)
    for (i=10;i<l+10;i++)
        p->data[l+j] ^= mult(p->data[l+9-i],invert(alog[i]^alog[j]));
```

/* done re-encoding, proceed with decode */

```
wb(p);
printf("degW %d\n",degW);
if (chien(p)) correct(p);
p->length -= 10;
}
```

/* wb() performs the Welch-Berlekamp iteration */

```
wb(p)
struct packet *p;
{
    /* initialize welch-berlekamp */
    register short j,k,lim;
    d=0;
    for (j=0;j<6;j++) w[j]=m[j]=n[j]=v[j]=0;
    w[0]=m[0]=1;
```

/* welch-berlekamp iteration */

```
for (j=0;j<10;j++) {
    /* set for-loop limit for polynomial operations */
    lim = ((j>4) ? 5 : j+1);
    /* get remainder from packet */
    s = p->data[(p->length - 10) + j];
    printf("remainder: %d\n",s);
    /* evaluate a and b */
    a = b = 0;
```

```

for (k=0;k<=lim;k++) {
    a ^= mult(x[j][k],n[k]) ^ mult(s,mult(x[j][k],w[k]));
    b ^= mult(x[j][k],m[k]) ^ mult(s,mult(x[j][k],v[k]));
}

/* test d, a, b and do a 4-way branch */

if ((d<=0) && !a) {

    /* update M, V */
    for (k=lim;k>0;k--) {
        m[k] = m[k-1] ^ mult(x[j][1],m[k]);
        v[k] = v[k-1] ^ mult(x[j][1],v[k]);
    }
    m[0] = mult(x[j][1],m[0]);
    v[0] = mult(x[j][1],v[0]);

    d--;
    continue;
}

if ((d>0) && !b) {

    /* update N, W */
    for (k=lim;k>0;k--) {
        n[k] = n[k-1] ^ mult(x[j][1],n[k]);
        w[k] = w[k-1] ^ mult(x[j][1],w[k]);
    }
    n[0] = mult(x[j][1],n[0]);
    w[0] = mult(x[j][1],w[0]);

    d++;
    continue;
}

if ((d<=0) && a) {

    /* update M, V */
    for (k=0;k<=lim;k++) {
        m[k] = mult(a,m[k]) ^ mult(b,n[k]);
        v[k] = mult(a,v[k]) ^ mult(b,w[k]);
    }

    /* update N, W */
    for (k=lim;k>0;k--) {
}

```

```

n[k] = n[k-1] ^ mult(x[j][1],n[k]);
w[k] = w[k-1] ^ mult(x[j][1],w[k]);
}
n[0] = mult(x[j][1],n[0]);
w[0] = mult(x[j][1],w[0]);

d++;
continue;
}

if ((d>0) && b) {

/* update N, W */
for (k=0;k<=lim;k++) {
    n[k] = mult(b,n[k]) ^ mult(a,m[k]);
    w[k] = mult(b,w[k]) ^ mult(a,v[k]);
}

/* update M, V */
for (k=lim;k>0;k--) {
    m[k] = m[k-1] ^ mult(x[j][1],m[k]);
    v[k] = v[k-1] ^ mult(x[j][1],v[k]);
}
m[0] = mult(x[j][1],m[0]);
v[0] = mult(x[j][1],v[0]);

d--;
continue;
}

/* done with W-B iteration; must set degW to degree of w[] */

degW=6;
while (!w[(degW--)-1]);
}

/* value() is called by chien with an argument which is the
   alog of a message error location. It returns the error value
   N(alog[l]) / W'(alog[l]) */

unsigned char
value(l)
short l;

```

```

{
unsigned char num,den,t,a;
t = (a = alog[1]);
num = n[0];
num ^= mult(n[1],t);
t = mult(t,a);
num ^= mult(n[2],t);
t = mult(t,a);
num ^= mult(n[3],t);
t = mult(t,a);
num ^= mult(n[4],t);
den = w[1];
t = mult(a,a);
den ^= mult(w[3],t);
t = mult(t,t);
den ^= mult(w[5],t);
return(mult(num,invert(den)));
}

/* chien searches for roots of w, updating the arrays
loc[] and val[]. chien returns 1 if decodable, in
which case loc[] and val[] have NumErrs location/value
pairs; chien returns 0 if undecodable */

chien(p)
struct packet *p;
{
unsigned short sum,w0,w1,w2,w3,w4,w5;
short l,lim;
short dtmp,numerrs;

printf("d: %d\n\n",d);

if (d>0) return(0); /* wb exited undecodable */

dtmp = degW;

/* first search for check roots of w, starting by initializing
w0, w1, w2, w3, w4, w5 for location alog[-1] = alog[254]. When a
root is found we only decrement dtmp */

#define A254_1 0x96
#define A254_2 0x4b
#define A254_3 0xb3
#define A254_4 0xcf

```

```

#define A254_5 0xf1

w0 = w[0];
w1 = mult(w[1],A254_1);
w2 = mult(w[2],A254_2);
w3 = mult(w[3],A254_3);
w4 = mult(w[4],A254_4);
w5 = mult(w[5],A254_5);

for (l=0;l<10;l++) {
    sum = w0;
    sum ^= (w1 = mult(w1,alog[1]));
    sum ^= (w2 = mult(w2,alog[2]));
    sum ^= (w3 = mult(w3,alog[3]));
    sum ^= (w4 = mult(w4,alog[4]));
    sum ^= (w5 = mult(w5,alog[5]));
    if (!sum) dtmp--;
}

/* printf("degW less number of check errors: %d\n\n",dtmp); */

/* w0...w5 are now set for location alog[9], and we are
   ready to search for message roots starting at the end
   of the message */

lim = (p->length);
NumErrs = 0;

for (l=10;l<lim;l++) {
    sum = w0;
    sum ^= (w1 = mult(w1,alog[1]));
    sum ^= (w2 = mult(w2,alog[2]));
    sum ^= (w3 = mult(w3,alog[3]));
    sum ^= (w4 = mult(w4,alog[4]));
    sum ^= (w5 = mult(w5,alog[5]));
    if (!sum) {
        dtmp--;
        loc[NumErrs] = l;
        val[NumErrs++] = value(l);
    }
}

printf("dtmp at end of chien(): %d\n\n",dtmp);

return(dtmp == 0);

```

```
}
```

```
/* correct() fixes the errors in the message section of the
```

```
packet */
```

```
correct(p)
```

```
struct packet *p;
```

```
{
```

```
int i;
```

```
for (i=0;i<NumErrs;i++)
```

```
    p->data[(p->length) - loc[i] - 1] ^= val[i];
```

```
}
```

APPENDIX C

FEC-II CODE

(The contents of this section are (Effectivity 2) pending further implementation and testing of the proposed FEC code.)

10. Scope. This appendix is not a mandatory part of the standard. The information it contains is intended for guidance only.

20. Applicable documents. This section is not applicable to this appendix.

30. FEC-II Code. FEC-II encoding applies both Reed-Solomon and BCH coding for operation in high error environments. Further, FEC-II encodes a single datagram or section of a datagram into a group of "fragments," or small packets of 12 bytes each. The BCH coding protects each fragment, and using the Reed-Solomon coding, up to eight fragments may be lost in transmission while still allowing the receiver to recover the datagrams.

(The term fragment used in this section, is unrelated to the fragments defined by the Internet Protocol.)

The individual fragments shall be encapsulated by the data link layer, generally either as a SLIP frame as specified in 5.4.3.2, or as an HDLC frame as specified in 5.4.3.1. If a FEC-II fragment is encapsulated as an HDLC frame, the 2-byte HDLC frame opening sequence, as well as the 2-byte CRC, shall not be included in the frame.

Datagrams up to length 382 bytes may be encoded by the FEC-II coding process. The first step shall be to represent the datagram by either one or two "sections." If the datagram is of length L, where L is 191 bytes or less, it shall be represented by a single "section" containing L + one bytes as follows: an initial byte containing the byte count L, followed by the L bytes contained in the datagram. If the datagram is of length L, where L is 192 through 382 bytes inclusive, it shall be represented by two sections. The first section shall consist of an initial byte with value 255 (decimal), followed by the first 191 bytes contained in the datagram. The second section shall consist of an initial byte with value (L - 191), followed by the remaining bytes of the datagram.

First we will specify the format of a single fragment, and then describe how the input bytes used to form the fragments are derived from the unencoded section.

Each 12-byte fragment is formed from an 8-byte sub-block, which is made up of the "input bytes" on figure C-1.

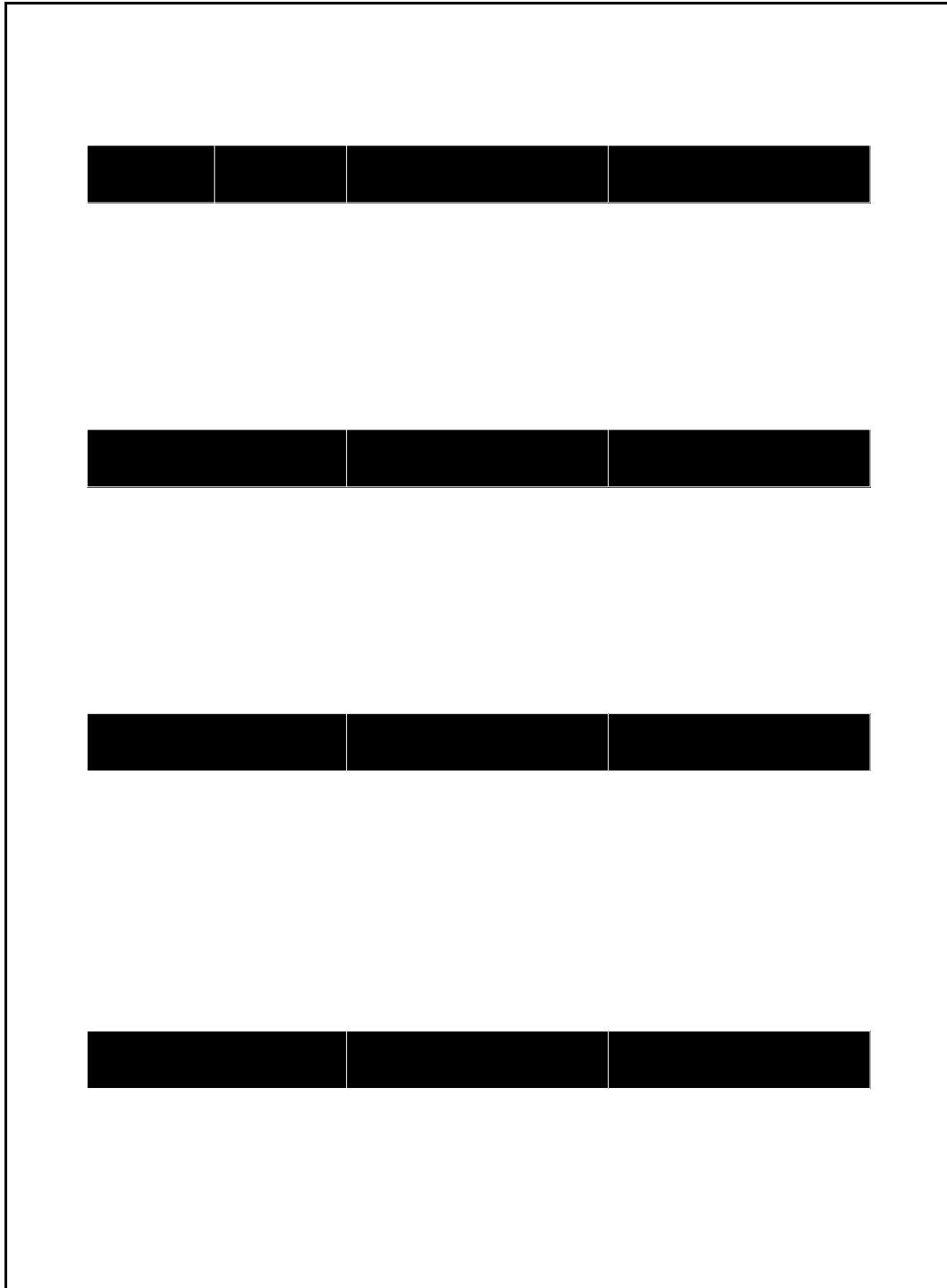


FIGURE C-1. FEC-II fragment format.

The origin of the 3-bit Modified Count Code and 5-bit sequence number will be described shortly. The 24-bit BCH redundancy field is formed using the following polynomial:

$$x^{24} + x^{23} + x^{21} + x^{20} + x^{19} + x^{17} + x^{16} + x^{15} + x^{13} + x^8 + x^7 + x^5 + x^4 + x^2 + 1$$

To calculate the BCH redundancy, feed the following bits, in the order shown, into a feedback shift register, initialized with the hexadecimal value 0x0000FF, and wired according to the above polynomial:

Modified Count Code bits 2 through 0
 Sequence Number bits 4 through 0
 IB0 bits 7 through 0
 IB1 bits 7 through 0
 .
 .
 .
 IB7 bits 7 through 0

The above describes how to form a fragment given the input bytes, modified count code, and sequence number. We now describe how the input bytes, modified count codes, and sequence numbers are derived.

First, the unencoded section is split into a sequence of 8-byte sub-blocks. Eight more 8-byte sub-blocks containing Reed-Solomon redundancy are then created. Each sub-block contains bytes numbered 0 through 7.

Eight choices exist for the number of message sub-blocks used to represent a datagram, as determined by a 3-bit count code in table C-I.

TABLE C-I. Count codes for different length sections.

Count Code	Number of Message Sub-blocks	Sequence Numbers of Message Sub-blocks	Nominal Section Length (bytes)
0	4	0-3	32
1	6	0-5	48
2	8	0-7	64
3	10	0-9	80
4	13	0-12	104

5	16	0-15	128
6	19	0-18	152
7	24	0-23	192

The eight Reed-Solomon Redundancy Sub-blocks always have sequence numbers 24 through 31.

The 3-bit Modified Count Code inserted in the fragment is the bitwise-exclusive-OR formed from the above Count Code, and a residue computed from the remaining data in the fragment. This residue is defined as follows (using binary arithmetic):

$$Residue = \left\{ (sequence\ number\ mod\ 9) + \sum_{i=0}^7 (Input\ byte\ i)\ mod\ 9 \right\} \mod 8$$

Each byte in a Reed-Solomon Redundancy Sub-block with sequence number j is computed from the correspondingly-numbered bytes M_i in each of the message sub-blocks as follows:

$$C_{T(j)} = \sum_i \frac{M_i}{a^{T(i)} = a^{T(j)}} , \quad 24 \leq j \leq 31$$

where i ranges over the message sub-block sequence numbers, and the correspondence between sequence numbers and locations within the Reed-Solomon codeword is:

x	T[x]
24	0
25	1
26	2
27	3
28	4
29	5
30	6
31	7
0	10
1	11
2	12
.	.
.	.
.	.
23	33

(The expression above uses Galois Field arithmetic as described in 5.4.2.1 for the FEC-I code.)

Presently, FEC-II encoding is not specified for datagrams whose unencoded length is greater than 382 bytes. Should a FEC-II encoder be presented with such a datagram, the correct action is to transmit it without any encoding.

Because the FEC-II encoding process has the effect of adding zero-fill to a datagram to achieve a standard length, a receiving system must determine the actual length of the original datagram. It does this by examining the length field in the Internet Protocol (IP) header, or, in the case where header abbreviation is used, the length field in the abbreviated header.